HAL
open science

Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur, Bernard Botella, Arnaud Gotlieb

To cite this version:

Florence Charreteur, Bernard Botella, Arnaud Gotlieb. Modelling Dynamic Memory Management in Constraint-Based Testing. Journal of Systems and Software, 2009, 82 (11), pp.1755-1766. 10.1016/j.jss.2009.06.029 . inria-00540270

HAL Id: inria-00540270

https://inria.hal.science/inria-00540270

Submitted on 26 Nov 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Modelling Dynamic Memory Management in Constraint-Based Testing

Florence Charreteur[1]
University of Rennes 1
Campus de Beaulieu
35042 RENNES – France
Florence.Charreteur@irisa.fr

Bernard Botella
CEA LIST
Saclay
91 191 Gif SurYvette – France
Bernard.Botella@cea.fr

Arnaud Gotlieb
INRIA
Campus de Beaulieu
35042 RENNES – France
Arnaud.Gotlieb@irisa.fr

Abstract

Constraint-Based Testing (CBT) is the process of generating test cases against a testing objective by using constraint solving techniques. When programs contain dynamic memory allocation and loops, constraint reasoning becomes challenging as new variables and new constraints should be created during the test data generation process. In this paper, we address this problem by proposing a new constraint model of C programs based on operators that model dynamic memory management. These operators apply powerful deduction rules on abstract states of the memory enhancing so the constraint reasoning process. This allows to automatically generate test data respecting complex coverage objectives. We illustrate our approach on a well-known difficult example program that contains dynamic memory allocation/deallocation, structures and loops. We describe our implementation and provide preliminary experimental results on this example that show the highly deductive potential of the approach.

Keywords: *Software Testing*, *Constraint-Based Testing, Automatic Test Data Generation, Dynamic structures*

1. Introduction

By increasing our confidence in the quality of software, testing techniques play a prevalent role in the verification process of software systems. However, software testing remains an expensive task in the development process and one of the main challenges concerns its possible automation. Since the seminal work of Offut and De Millo in the context of mutation testing [1], much attention has been devoted to the use of constraint solving techniques in the automation of software testing (*Constraint-Based Testing*). In [2], Dick and Faivre proposed to exploit a VDM model to generate automatically test cases for partition testing while Marre proposed in [3] to exploit Constraint Logic Programming to generate test cases from an algebraic specification. In 1998, Gotlieb, Botella and Rueher proposed using constraint propagation techniques to generate test data for the structural coverage of C programs [4]. This work resulted in the development of the INKA tool which was the first to address C programs containing pointers (but without dynamic allocation) and floating-point variables [5,6,7]. Meudec followed a similar path with ATGen, a software test data generator based on Constraint Logic Programming for ADA programs [8] and more recently, Williams introduced a new test data generation method in a tool called PathCrawler, based on the on-the-fly generation of paths in C programs [9]. This method was independently discovered by Godefroid and Sen in the DART and CUTE approaches [10,11]. Note that Constraint-Based Testing methods cover several applications area including hardware

[1] Corresponding author – Florence Charreteur – Tel/Fax : +33 299 842 2 86/+33 299 847 171

verification [12,13], test data generation for structural testing [4-11], functional testing [1,2], counter-example generation [14,15], and software verification [16].

Constraint-Based Testing (CBT) is a two-stage process consisting in generating test data against a user-selected testing objective. The first stage is a **constraint generation step** aiming at extracting a constraint system from the source code of a program under test and a selected testing objective, while the second stage is a **constraint solving step** consisting in trying to solve the constraint system in order to get a test data that satisfies the testing objective. For example, by selecting a statement within the program under test, we get a constraint system that characterizes a subdomain of the program input domain and solving this constraint system leads to generate a test data in the subdomain, on which the statement is executed. Such an approach has been called goal-oriented test data generation by Fergusson and Korel [17] as opposed to path-oriented test data generation which consists to select first a path within the program before trying to generate a test data that activates the corresponding path. When the constraint system has no solution, then the testing objective is unreacheable. For goal-oriented test data generation, it means that the corresponding statement or decision is unreacheable. Such deductions are usually outside of the scope of any path-oriented test data generators as soon as a loop is present in the program because the path selection process is possibly endless in this case. Note however that showing the unsatisfiability of a constraint system is undecideable in the general case[2] and then usually some unreacheable elements may remain undetected even for goal-oriented approaches.

This paper addresses the problem of dynamic memory management in a goal-oriented test data generation approach. Dynamic structures are heap-based data structures built during the execution of the program. CBT approaches cannot easily handle these structures, as their exact shape cannot be completely known at compile time. One usually resort either to use an approximation of the structure shape by static analyses or to use dynamic analysis. In the first case, one cannot make exact deductions about the states of the memory manipulated by the program as the program semantics has been approximated, while in the second case, deductions can only be performed on some program executions.

1.1 A motivating example

Let us illustrate this problem on a non-trivial example that belongs to the folklore of C pointer problems [18]. The Josephus program, shown in Fig. 1, is a decimation problem and relates to a Historical situation where Jewish rebels were surrounded by Romans and decided to commit to suicide: they lined up in a circle and systematically killed every other one, going around and around, until only one rebel is left. The program contains two successive loops: the first one builds a circular simple-linked list of n nodes, while the second eliminates nodes at position m until only a single node remains in the list. Thus, this program contains dynamic memory allocation/deallocation within loops, which is the main technical difficulty concerning dynamic structures in CBT approaches. Indeed, an interesting testing objective for the Josephus program is to find values for m and n such that the second loop (**while 2**) is unrolled at least forty times, as this number corresponds to the Ancient problem[3]. Such complex testing objectives are frequent in practice, as interesting states of a system often result from complex control flow. Note also that such states are usually difficult to reach by hand even for experienced validation engineers [19], making automation a real challenge. For CBT tools, finding values for m and n such that the second loop (**while 2**) is unrolled at least forty times

[2] As a consequence of the 1970's Matiyasevitch result on the undecideability of the Tenth problem of Hilbert

[3] In the original ancient Josephus's decimation problem, there were 40 people killed, letting the rebel alive at position 31.

involves either complex static analyses such as structure sharing analysis and pointer aliasing analysis or dynamic analysis (based on program executions). A static analyzer would typically ignore the backward pointer toward the first element of the list, while a dynamic analyzer would have very few chances to satisfy our testing objective as they are usually based on initial random draws. On the contrary, thanks to our operators that model the dynamic memory management, our system automatically deduces that n, the length of the initial list should be 41 while the remaining element at the end of the process is 31 when m=3. This corresponds exactly to the expected solution of the original problem (the surviving rebel will be the one positioned at the 31th position in the circle).

```
typedef struct node *link;
struct node { int key ; link next;};

int f(int n,int m){
1. int i; link t,x ;
2. t=(link)malloc(sizeof(struct node));
3. t->key = 1;
4. x = t;
5. i = 2;
6. while( i <= n ){          //while1
7.   t->next=
       (link)malloc(sizeof(struct node));
8.   t = t->next ;
9.   t->key = i;
10.  i++;}
11.t->next = x ;
12.while( t != t->next){  //while 2
13.  i = 1;
14.  while( i <= m-1){   //while 3
15.    t = t->next ;
16.    i++;}
17.  x = t->next ;
18.  t->next = (t->next)->next ;
19.  free(x);}
```

Figure 1. Josephus program

1.2 Contributions

The main contribution of the paper is the design of several constraint operators that model memory allocation, deallocation, accesses and updates. These operators are equipped with deduction rules useful to perform constraint reasoning on imperative programs, as required by CBT tools. We present each operator under the form of finite state machine that interacts with the constraint solver. Such a presentation is advantageous as it makes clearer the implementation of these operators. Another contribution of the paper is the design of a complete goal-oriented test data generation method for C programs containing dynamically allocated structure. Our approach can deal with circular lists as well as any memory shapes that may include backward pointers. We are not aware of any other symbolic approaches able to deal with these data structures (see section 7). We implemented our constraint operators within the test data generator INKA [7] and got preliminary experimental results that show the potential of the constraint operators to reason about program with dynamic memory management.

1.3 Outline of the paper

Section 2 recalls the necessary background on constraint solving over finite domains and states some notations and restrictions. Section 3 introduces our overall goal-oriented constraint-based test data generation technique. Section 4 details the abstract memory model we use to deal with dynamically allocated structures. Section 5 presents the deduction rules exploited in the constraint operators under the form of abstract state machines; Section 6 gives our preliminary experimental results while section 7 discusses related works. Finally, Section 8 concludes and draws some perspectives to this work.

2. Background

2.1 Constraint solving over finite domains

Our approach is based on constraint solving over finite domains. In this framework, a finite domain is associated to each variable and a solution to the constraint system is a valuation of the variables within their domains that satisfy each constraint. Primitive constraints are built over variables, domains, arithmetical operators in {+,-,*,\,…} and relations {>,≥,=,≠,≤,<} while non-primitive constraints include user-defined constraints and constraint operators that express high-level relation between other constraints. In this paper, we define constraint operators that model dynamic memory allocation, deallocation, accesses and updates. These constraints apply deduction rules as any other constraint of a finite domain constraint solver.

Two interleaved processes intervene in the solving process of a finite domain constraint system: constraint propagation and variable labeling.

2.1.1 Constraint propagation

Initially, the constraints are added to a main queue and fall into in an ***evaluation*** state. Each constraint of the queue is considered one-by-one by the constraint propagation algorithm. The algorithm exploits each constraint to filter out the inconsistent values from the domain of the variables. When the domains of all variables of the constraint have been pruned, the constraint falls into the ***suspended*** state. When the domain of a variable is pruned, other constraints that involve this variable are reintroduced into the queue. In this case, these *suspended* constraints are woken up and return in the *evaluation* state. If the domain of at least one variable becomes empty, the constraint ***fails***: the constraint system is unsatisfiable. If the constraint succeeds meaning that each of the tuples from the current domains are compatible with the constraint, then it falls in the ***entailed*** state. In this case, the constraint is removed and is not considered anymore in the process since it is no more useful. When no more reduction is possible, the queue becomes empty and the constraint propagation ends.

2.1.2 Variable labeling

When the constraint propagation ends, enumeration on the possible values from the domains is usually required to get a solution. The labeling procedure tries to give a value to every variable one by one. When a value is chosen from the domain of a variable, the constraint propagation is re-run to prune the domains of other variables with the current hypothesis. If a contradiction appears during the resolution process, the procedure backtracks to other possible values. The process stops when a value is assigned to each variable.

2.2 Notations, syntax and restrictions

2.2.1 Notations

In the rest of paper, we will use capitals, possibly subscripted, to denote finite domains variables and variable of the constraint model (such as memory for example). On the contrary, we will use lower-case letters to denote program variable and will use a special operator, noted @, to denote addresses of the memory.

2.2.2 Syntax and restrictions

```
program      ::= { statement* }
statement    ::=
    assignment
  | malloc(type_size)        %memory allocation
  | free(expr)               %memory deallocation
  | if( expr ) { statement* } else { statement* }
  | while( expr ) { statement* }
assignment   ::=
    var = expr  |  *var = expr  |  var->f = expr

expr ::=
    cte | var | &var | *var | var->f | expr + expr | …
```

Figure 2. Syntax of the pointer language

For the sake of clarity, we confine ourselves to a small C-like language (whose an excerpt of the grammar is given in Fig. 2) that includes pointers assignment and dereferencing, structures management, memory allocation and deallocation. This language is powerful enough to express any computable function and possesses all the necessary features to deal with dynamic allocated structures. However, it also presents a lot of restrictions w.r.t. the C language. In particular, it does not allow unstructured code, type casting, unconstrained pointer arithmetic, volatile variables, function calls, etc. In this paper, we focus on the problem of dynamic memory management within our constraint-based test data generation method, hence we will only detail the constraint model for the operators on memory and will left apart the rest. Similarly, we will only give an overview of the complete test data generation method. The interested reader can consult ref [4] to see precisely how we deal with control structures (conditionals, loops), ref [5] to understand the problems of pointer aliasing in constraint-based approaches, and ref [6] to understand how we deal with floating-point computations.

3. Goal-oriented test data generation based on constraint solving

3.1 Constraint generation

As said previously, CBT involves two processes: constraint generation and constraint solving. In our framework, the constraint generation step involves the translation of a program under test into a constraint model over finite domain variables and memory variables. This constraint model can be seen as a relational formulae over the input memory state of the program and its output memory state. Note that such memory states can contain unknown references such as input pointer formal parameters or unknown integer variables. In fact, there is no strong hypothesis on the context of call and each C function is treated in isolation in our model. Moreover, our goal-oriented approach prevents using path expression to remove ambiguities. As soon as a conditional or a loop is encountered, our model can contain unknowns due to the non-determinism of the control flow. Consequently, we need a constraint model able to deal with unknown parts of the memory. We get this model by a systematic and inductive translation of statements into constraints. We will describe this model on a simple but illustrative example. Consider the following sequence of statements:

```
        C code                                  Constraint model

 a.     i = i+1          ------------>     load_elementt(M1, @i, I1)
                                          I2 = I1 + 1
                                          store_element(M1, M2, @i, I2)

 b.     if( i > 10 )     ------------→      load_element(M2, @i, I3)
                                           B1 = (I3 > 10)
                                            ite(B1,
 c.        { p = &i;}                             store_element(M2, M3, @p, @i)and M4 =
M3,
                                               M4 = M2
                                                )

 d.    *p = *p + 5       ------------>    load_element(M4, @p, P)
                                          load_element(M4, P, DP1)
                                         DP2 = DP1 + 5
                                          store_element(M4, M5, P, DP2)

 e.    if( i < 16 ) …    ------------→   load_element(M5, @i, I4)
                                          B2 = (I4 < 16)
                                          ite(B2, …
```

The C code presents an aliasing problem at statement d as we ignore whether p points to i or not at this point (suppose for example that i and p are input formal parameters of the program under test). Statement `a` is translated into three independent constraints. The first one states a relation between the (abstract) memory `M1` and the variable value `I1` that is associated to program variable `i` through its reference `@i`. This relation holds anytime an access to program variable `i` is made, but depends on the current state of the memory at a given point of the program. The second constraint states a relation between two finite domain variables (`I1` and `I2`) where `I2` is a new fresh variable. Our constraint model does not require variables to be declared. Finally, the third constraint links memories `M1` and `M2` (two states of the same abstract memory), and variable `I2`: the new state of `M2` should be the update of the state `M1` with `I2` at reference `@i`. This decomposition shares similarities with a classic 3-

address code that can be found in many compilers. Statement `b` is a conditional and we use an auxiliary boolean variable `B1` to associate with the truth value of the decision `i > 10`. The conditional itself is translated to a special constraint operator in our constraint model noted `ite(B1, Then, Else)`. Loops are equally treated with a special constraint operator called `w`, see [4,7] for more details. Note that, when no deduction is possible on the path that should be followed, the conditional operator makes the union of the two possible memory states. In our example, M2 and M3 joint in the memory state M4. M3 is the memory state obtained after interpretation of the constraints of the Then-part of the conditional while M2 is just the memory available if the Else-part is taken (no statement in this example). This way of interpreting conditional has much to do with the Static Single Assignment form of imperative program [4]. Statement `c` is translated into a single `store_element` constraint that links memory `M2` and `M3`. Statement `d` is translated into four constraints. The two first permit to access to the value stored in memory `M4` at reference `*p`, while the two latter permit to store the result of expression to memory `M5`. The need for `load_element` to be a constraint relation (as opposed to a function) is clearly stated at statement `e`. As the value of `p` is determined by the control flow, we have to get it at the memory state `M5` which is unknown or only partially known before having selected a path through the conditional. One can see here how pointer aliasing is handled through the use of special constraint operators. This also appeals for powerful deduction rules in case some information is available on memory states. As a result, this constraint model allows us to implement goal-oriented test data generation in a constraint-based approach.

3.2 Constraint solving

The constraint solving step aims at finding an input memory state that satisfies a given testing objective. Consider the objective of generating a test data that reaches the Then-part of statement `e`. By using the control dependencies within the program, this testing objective is directly translated into the constraint `B2 = 1`, forcing I4 to be strictly less than 16 and so constraining the memory variable `M5`. By applying the deduction rules of each constraint operator of the model in a constraint propagation algorithm, our system makes the non trivial deduction that the Then-part of statement `b` cannot be executed. In fact, if the program variable `i` at statement `b` is strictly greater than 10 then `*p` and `i` would be aliased at statement `d`, and the value of `i` would be greater than 16 at statement `e`, which is contradictory with the testing objective. As a consequence, our system deduces that the value of `i` at the beginning of the program has to be less than 9. Thanks to the relational view of each statement, constraint reasoning is possible and deduces interesting facts about the input state of the program in order to satisfy the selected testing objective. Test data generation is obtained just by launching a labelling search that chooses i to be zeroed in the input state of the memory.

It is worth noticing that our constraint reasoning did not exclude the possibility that p points to i in the input state of the memory. Such input aliasing relationship depends on the context of call.

4. Description of an abstract memory

In our system, we express C operations over the memory with relations over two (abstract) memory states. The first memory state represents the memory before statement execution while the second memory state represents the memory after statement execution. An **abstract memory** in our constraint system is an abstract model of the physical memory at a given program point and contains information known *at a given step of the resolution process of the overall constraint system*. Figure 3 describes the content of an abstract memory. m : it contains four elements: struct(m), tab_i(m), tab_p(m), and closed(m) that are described in the following subsections.

Description of sets :
MEM: set of abstract memories
$IDENT$: set of locations in abstract memories
VAR_i: set of abstract integer variables
VAR_p: set of abstract pointer variables
VAR_s : set of abstract structures
$TYPE_s$: set of structured types
$NAME_f$:set of names of fields
$TTPE$: set of types. $Type = \{integer, pointer\} \cup Type_s$

$\forall m \in MEM : m =< tab_i(m), tab_p(m), struct(m), closed(m) >$
with: $tab_i(m) \in \mathcal{P}(\{< ident, V_i > | ident \in IDENT, V_i \in VAR_i\})$
$tab_p(m) \in \mathcal{P}(\{< ident, V_p > | ident \in IDENT, V_p \in VAR_p\})$
$struct(m) \in \mathcal{P}(\{< type, V_s > | type \in TYPE_s, V_s \in VAR_s\})$
$closed(m) \in true, false$

$s\_@ : \ VAR_s \rightarrow \mathcal{P}(IDENT)$ % set of program locations
$access\_f : IDENT \times NAME_f \rightarrow IDENT$ % access to a field
$dom_i : VAR_i \rightarrow \mathcal{P}(\mathbb{N})$
$dom_p : VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL) \cup \{all\}$
$ndom_p VAR_p \rightarrow \mathcal{P}(IDENT \cup NULL)$
$type_p : VAR_p \rightarrow TYPE$

Figure 3. Abstract memory

In order to facilitate the understanding, we provide an example of abstract memory. Figure 4 shows the abstract memory state m obtained before statement 11 of the Josephus program after two iterations of the first loop.

$tab_i(m)$

Ident	Var_i	dom_i
m	V_i1	Inf..sup
n	V_i2	2
i	V_i3	3
n(2).key	V_i4	1
n(7.1).key	V_i5	2
n(7.2).key	V_i6	3

$tab_p(m)$

Ident	Var_p	dom_p	non_dom_p	$type_p$
t	V_p1	{n(7.2)}	empty	node
x	V_p2	{n(2)}	empty	node
n(2).next	V_p3	{n(7.1)}	empty	node
n(7.1).next	V_p4	{n(7.2)}	empty	node
n(7.2).next	V_p5	all	empty	node

struct(m)=<node, S_{node} >

Var	s_@
S_{node}	{ n(2), n(7.1).n(7.2)}

closed(m)=true

Figure 4. Abstract memory after the statement 11 of the Josephus program, at the end of the constraint propagation when the first loop is enrolled twice

4.1. Structures dynamic allocations: struct(m)

For an abstract memory m at a given step of the solving process, information about known (statical or dynamical) allocation of structures is stored in struct(m). In a memory, a variable is associated to each structure type definition. In figure 4, the variable S_{node} is associated to the type node. A function called s_@ gives the set of anonymous program locations associated to this variable. On the model, we represent each abstract memory location with the term ident. An abstract memory location can be anonymous in the case of dynamic allocation. In figure 4 where there are three dynamically allocated structures, *{n(2), n(7.1), n(7.2)}* is the set of the anonymous program locations. For example, *n(7.2)* denotes the anonymous program location obtained by the execution of the statement 7 in the second iteration of the loop. The set given by s_@ can only be enlarged during the resolution process ot the constraint system. Function access_f associates to location loc of a structure and field name f, the complete name of the field location loc.f. For example, *access_f(n(1),next)* is *n(1).next.*

4.2. Basic types: $tab_i(m)$, $tab_p(m)$

Information about basic variables is memorized by pair ident-Var; where ident is an abstract memory location, and Var is a variable of type integer or pointer. On the model, the sets of integer and pointer variables are noted respectively VAR_i and VAR_p. These pairs are stored in two data structures, called *tableaux*: $tab_i(m)$, $tab_p(m)$. The number of pairs contained in the *tableaux* represent the known integer or pointer locations and therefore can only increase or remain the same during the resolution process of the constraint system..

4.2.1. Integer variables. For abstract variables that represent integers, the function dom_i gives the set of possible values at a given step of the resolution. For example, the abstract memory location i in $tab_i(m)$, has value 3 in abstract memory of figure 4.

4.2.2. Pointer variables. For abstract variables that represent pointers, our model provides two functions dom_p and $ndom_p$. dom_p returns the set of possible abstract memory locations (symbolic names or anonymous program locations) for the pointer while $ndom_p$ returns the set of memory locations that cannot be pointed by the pointer. For example, on a condition such as (p==&a||p==&b), we get $dom_p(P)=\{@a,@b\}$ where @a (resp. @b) denotes the address of a (resp. b) in the abstract memory model. $ndom_p$ is usefull in the case when dom_p=all. dom_p=all means that p can point to any location in the memory except the locations contained in $ndom_p$. On a condition such as (p!=&a) there are two possible changes in dom_p and $ndom_p$. If dom_p=all, then @a is added to $ndom_p$. Otherwise, @a is removed from dom_p. These deductions on pointer domains are expressed by the notation $P \neq @a$ in the description of the operators in section 5. If $dom_p(P)$ contains a single value v, the element pointed by P is definitely known. It is noted *P=v*.
For example, *Vp2* in figure 4 is simplified to *n(2)* as x points to the first dynamically allocated structure in statement 2.
The function $type_p$ returns the type of the pointed element. In figure 4, all the pointers point to a node structure. The number of elements in Dom_p can only decrease or remain the same during the resolution process of constraint system.

4.3. Closure of the memory: closed(m)

The last component of an abstract memory m is the predicate closed(m). It indicates whether all the elements of the *tableaux* and structures are defined or not in the current abstract memory. If closed(m) is false for an abstract memory at a step of the resolution process of the constraint system, it can become true later. If closed(m) is true at a step of the resolution process, then it remains true during all the process. The labeling process uses this predicate to force the input domain to contain only the data structures previously labeled. Without this predicate, labeling could invent new structures and values leading to a non-terminating process. Such status is of particular interest when one looks for the possible shapes of a dynamic structure during the labeling process as it strongly constrains the set of identifiers that can be pointed to by a pointer.

5. Constraint operators on abstract memories

To find a test datum to reach a given testing objective, the program under test is translated into a constraint system on abstract memories. This section details the deduction rules applied within the constraint operators that tackle with dynamic structures. Figure 5 shows the translation of some basic statements of our language into these constraint operators. M_{in} and M_{out} are the memories respectively before and after a given statement. All the elements of the memory M_{in} and M_{out} but the input parameters of the constraint operators are identical. Asterisks in figure 5 denote the operators that are detailed in the following. For other operators, we will only give an overview.

new(t,id)
(*) new_s(Struct(M_{in})(t), Struct(M_{out})(t),id)
new_fields($M_{in,}M_{out}$,t,id)

x=y->f
(*) load_element(Tab_p(M_{in}),y,Y)
(*) access_s(Struct(M_{in})(typeY),Y,f,V_p)
(*) load_element(Tab_t (M_{in}),Vp,Val)
(*) store_element(Tab_i(M_{in}), Tab_i(M_{out}),x,Val)

y->f=x
(*) load_element(Tab_p(M_{in})),y,Y)
(*) access_s(Struct(M_{in})(typeY),Y,f,V_p)
(*) load_element(Tab_t (M_{in}),x,X)
(*) store_element(Tab_i(M_{in}), Tab_i(M_{out}),Vp,X)

x=y
(*) load_element(Tab_t(M_{in}),y,Y)
(*) store_element(Tab_t(M_{in}), Tab_t(M_{out}),x,Y)

free(x,t)
(*) load_element(Tab_p(M_{in}),x,X),
(*) delete_s($Struc_t$(M_{in})(t), Struct(M_{out})(t),X)
delete_fields($M_{in,}M_{out}$,type,X,)

Figure 5. translation into operators

Statement **new(t,id)** generates two operators. The first one links the set of locations of type t between M_{in} and $M_{out,}$. It needs the identifier id of the location to add. The second constraint operator reserves place for the fields of the new structure. The statement **x=y->f** generates four operators. The first one loads the value of y. The second one gets a pointer Vp that points to the possible locations of y->f. Thanks to Vp, the third operator collects in Val the possible values for y->f. t in the notation Tab_t (M_{in}) means that the tableau from which the value is loaded depends on the type of y->f (integer or pointer). The fourth operator sets the domain of Val as domain of possible values for x. Similarly, statement **y->f=x** generates four operators. The first one loads the value of y. The second one gets a pointer Vp that points to the possible locations in memory that can store y->f. The third one loads the value of x. The fourth one affects the value of x to y->f. Statement **x=y** is expressed by two operators. The first one loads y. The second operator stores y in the location of x in the abstract memory. Statement **free(x,t)** generates three operators: the first one loads the value of the pointer x. The second one deletes the structure pointed by x in the memory while the third one deletes its fields.

We now turn on the description of these constraint operators: new_s for allocation, delete_s for deallocation, access_s for structure field access, store_element to store integer/pointer values in memory, and load_element to load values. The constraint operators for dynamic memory management have three roles: the propagation of the knowledge of the allocated locations in the memories, the filtering of the domains for pointers and integers values and the propagation of information about the closure of the memory.

5.1. Representation of a constraint operator

We propose to explain our constraint operators by using a simple model based on finite state machines. Figure 6 shows a generic state machine for constraint.

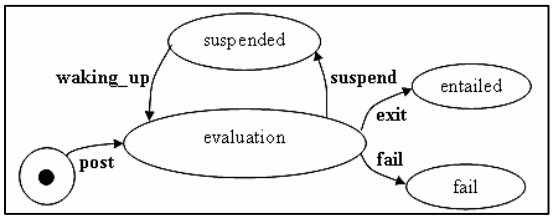


Figure 6. representation of a constraint operator

Our representation gives the possible states of a constraint: once posted, it can be in evaluation, suspended, entailed or in a failure state. Labels on the arrows describe the events that permit to switch from one state to another. For each operator, we describe five possible transitions: 1) post event: the operator is posted in the propagation queue ; 2) waking-up event: the operator is woken up by some additional information on the domain of its variables or on its status ; 3) suspend event: no more deduction rules can be exploited to prune the variation domains ; 4) exit event: the operator becomes entailed ; 5) fail event: some inconsistency has been detected which indicates failure of the current constraint system.

Figure 7 shows a C program that illustrate the interest of the deduction rules that we are going to present. In the following, we

```
    a= new(t,new(1)) ;
M0
    b= new(t,new(2)) ;
M1
    c= new(t,new(3)) ;
    a-> f =1;
    b-> f =2;
    c->f =3;
    if(cond1){
       p=a;
    }else{
       if(cond2){
           p=b;
       }else{p=c}}
    if(cond3){
M3
       free(p);
M4
    }else{
M5
       p->t=i;
M6
       j=p->t;}
    if(p!=a && b->t=6 && j>2){
```

Figure 7. program foo

will refer to this program implicitly by showing only the abstract state of the memories M0,..,M6.

5.2. The new_s operator

The operator new_s(S0,S1,id) is added whenever a structure of type t is dynamically allocated. Figure 8 shows the model of this operator that establishes a relation between S0, S1 and id. In the model, we suppose that

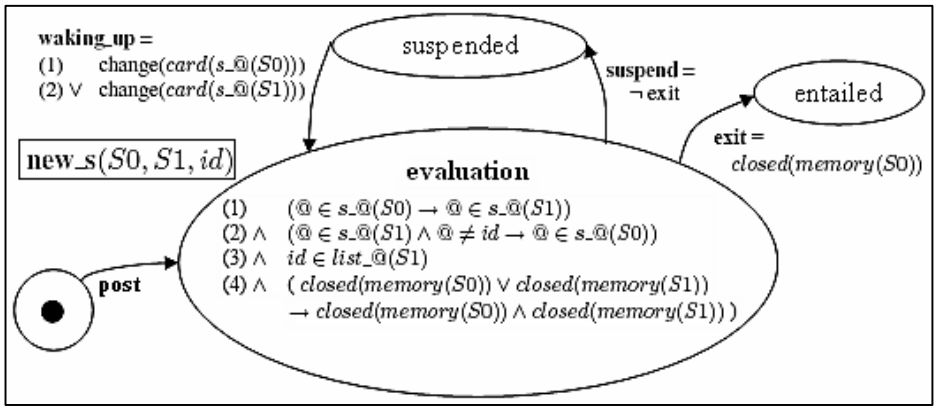


Figure 8. new_s operator

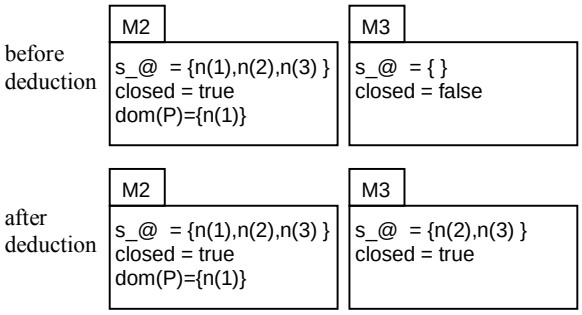
S0=struct(M_{in})(t), S1=struct(M_{out})(t), where struct(M)(t) denotes the variable associated with t in struct(M), and id is the identifier of the anonymous dynamic location. The operator is awoken when a location is added to the set of locations of S0 or S1.

If the operator is awoken, some new deductions can be performed. S0 and S1 should contain the same identifiers, except for id that is only in S1. Firstly, S1 should contain all the locations that are present in S0 (proposition 1) as well as the location id (proposition 3). Secondly, all the locations that are in S1, except id, should be in S0 (proposition 2). Moreover, if one abstract memory is closed, the second one should also be closed (proposition 4). Indeed, as new_s adds only one new location, the closure of M_{in} (resp. M_{out}) implies the closure of M_{out} (resp. M_{in}). Moreover, the operator succeeds as soon as M_{in} is closed (cf exit arrow), while it suspends otherwise.

5.3. The delete_s operator

The operator delete_s(S0,S1,X) is added whenever a structure of type t, pointed by X, is removed from the abstract memory. Figure 9 illustrates the delete_s operator, which maintains a relation between three parameters: S0=struct(M_{in})(t), S1=struct(M_{out})(t), and X.

The operator delete_s is woken up either when a location is added to the set of locations S0 or S1, or when the variation domain of pointer X is modified (for example, learning that X points to only one location). Such a modification is noted change(X) in our model. The relation maintains the fact that X should be non-null (proposition 1). As an illustration of the deduction rules, suppose the information on abstract memories linked by a delete_s operator shown below is available (memories before deduction).



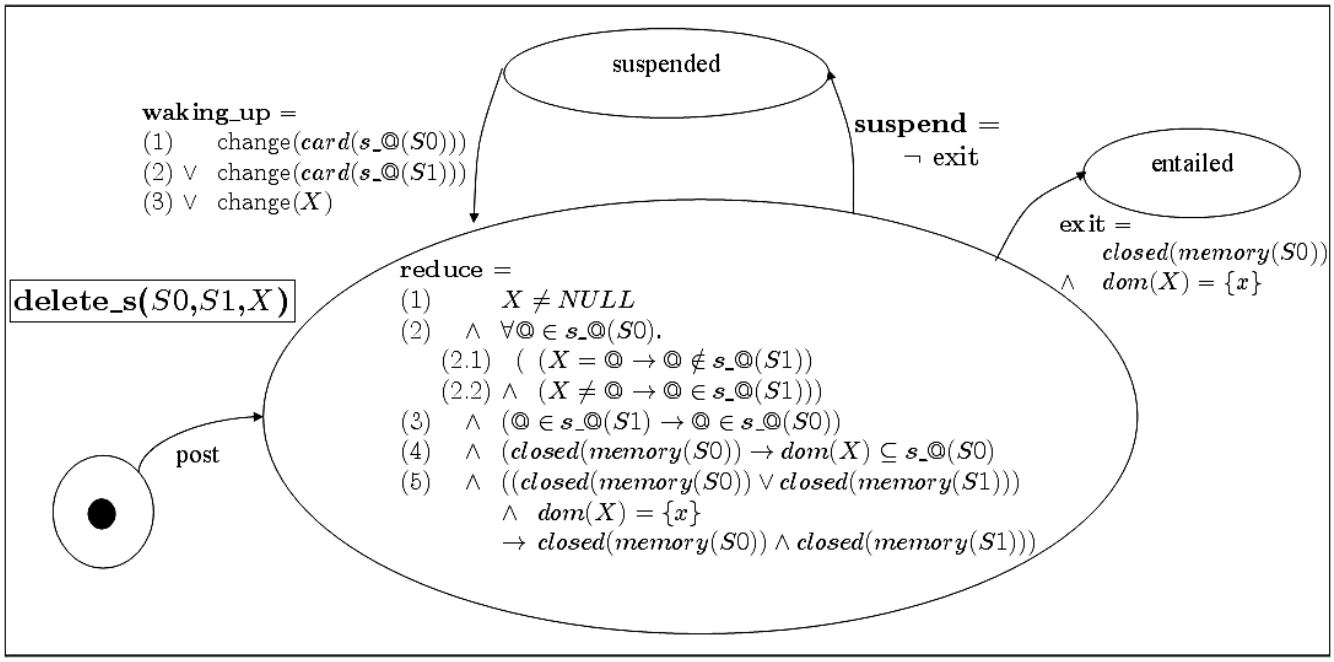


Figure 9. delete_s operator

Here, P points definitely to n(1) which is the deleted location. So, it should not appear in M3 (proposition 2.1) and other locations are not touched (proposition 2.2) and must appear in M3. Moreover, as M2 is closed and the location to delete is known, M3 is also closed (proposition 5) and the operator succeeds (exit arrow). We obtain the memories after deduction shown above. If the input memory is closed, then X can only point to a location contained in the set of addresses of S0 (proposition 4).

If there is no precise information on the pointed location or the available memory, then the operator is suspended.

5.4. The access_s operator

The operator access_s(S,X,f,Vp) is added when we access to a field of a structure of type t. Figure 10 illustrates this operator that maintains a relation between four parameters: S=struct(M)(t), X the pointer to the possible locations of the structure, f the field name and Vp the pointer to the possible locations of the field.

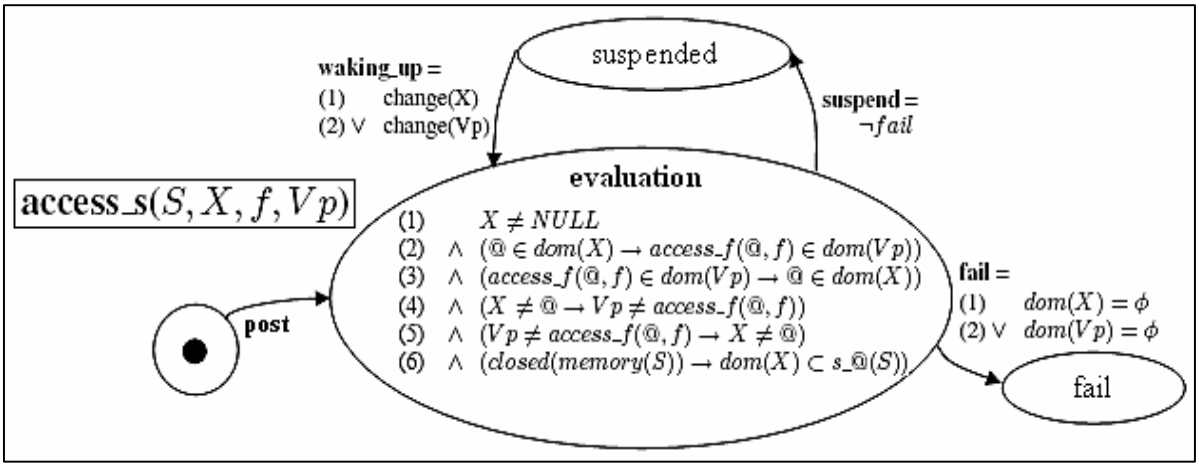


Figure 10. access_s operator

The operator access_s is awoken when we get more information about the values pointed by X or by Vp. X should not be null (proposition 1). Statement `p->t=i;` in figure 7 leads to add two constraint operators including access_s(struct(M4)(t),P,t,Vp). The following draw illustrates the deductions made by this operator. For each location of the domain of P, the relation maintains that Vp can point to the location associated with the field (proposition 2).

before deduction

M4

dom(P) ={n(2),n(3)}
dom(Vp)=...

after deduction

M4

dom(P) = {n(2),n(3)}
dom(Vp)={n(2)->t,n(3)->t}

In the following example, Vp can only point to n(2)->t or n(3)->t so P can only point to n(2) or n(3), n(1) is removed from its domain. (proposition 5).

before deduction

M4

dom(P) = {n(1),n(2),n(3)}
dom(Vp)= {n(2)->t,n(3)->t}

after deduction

M4

dom(P) = {n(2),n(3)}
dom(Vp)={n(2)->t,n(3)->t}

Proposition 6 gives complementary information: if the memory M is closed, all the possible locations pointed by X belong to the set of addresses of S. Indeed all the possible locations for a structure of type t are known and X points to such a structure.

During the resolution process, if the domain of a variable becomes empty, the constraint resolution process fails. Indeed, it means that there is no assignment for all the variables of the system such that all the constraints are satisfied. For the operator access_s, the only domains that can become empty are the domain of X and the domain of Vp (in this case X or Vp cannot point to any location anymore).

5.5. The store_element operator

The operator store_element(Tab(M_{in}),Tab(M_{out}),X,V) is added when the statement stores a value in memory at a given address. Figure 11 illustrates this operator that maintains a relation between X, Tab(M_{in}), Tab(M_{out}) and V. X is a pointer to a location that stores an integer or a pointer value, V is the value to store.

The store_element operator is awoken when 1) a pair <ident,Var> is added in one of the *tableaux* Tab(M_{in}) and Tab(M_{out}) (conditions 1 and 2) ; 2) when the domain of a variable in Tab(M_{in}) or Tab(M_{out}) changes (condition 3); 3) when the information about the pointer X changes; 4) when the domain of the value V to store changes.

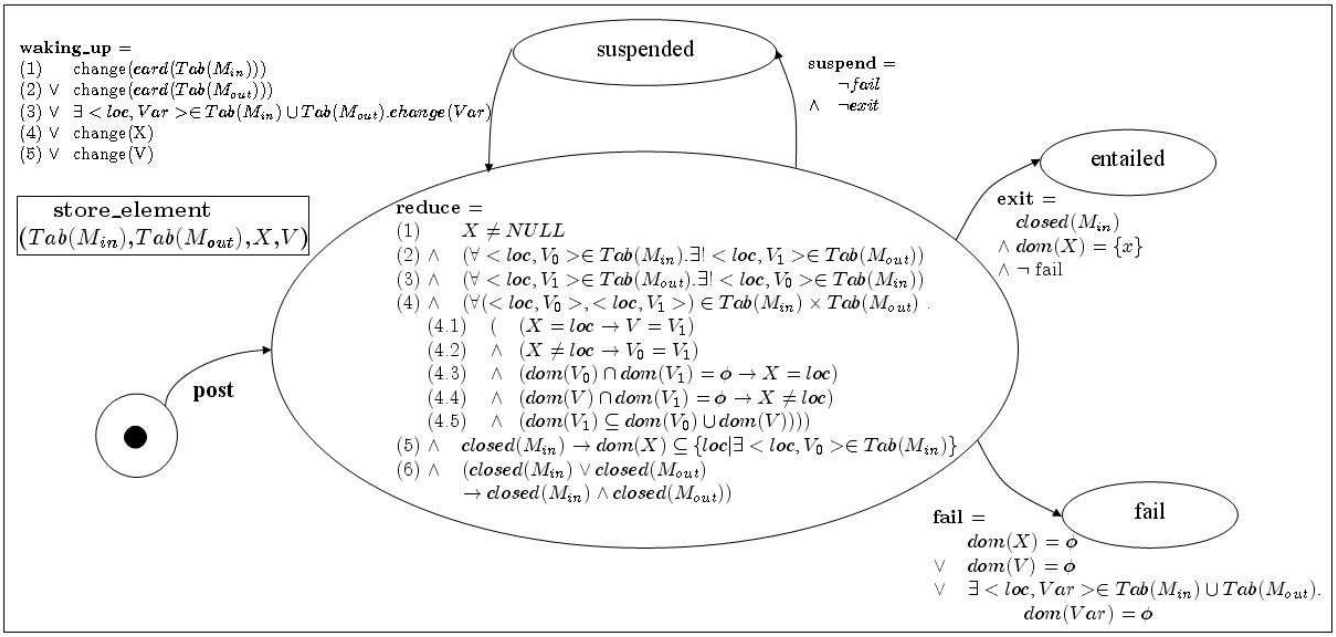
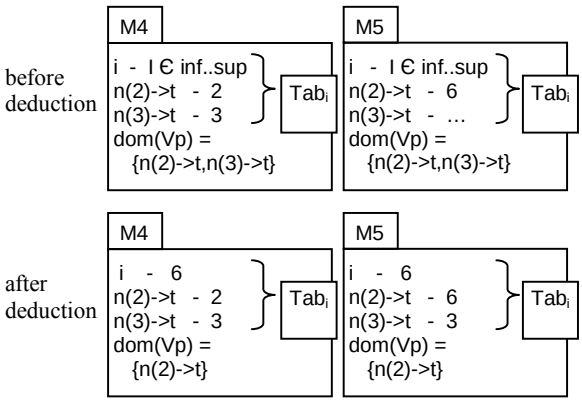


Figure 11. store_element operator

X should be non-null (proposition 1). Consider again statement `p->t=i;` in figure 7 and let Vp be a variable that points to the possible locations of p->t. The storage of the value of I in the location pointed by Vp is performed within the relation maintained by the store_element operator.

In the figure below, in the memories before deduction, as domains of n(2)->t before and after the storing statement are distinct the value of n(2)-> t is changed by the storing statement. It means that the value of I is stored in n(2)->t. As a consequence, Vp points to n(2)->t (proposition 4.3). dom(n(2)-> t) in M5 and dom(I) should be intersected in order to find the values of I and n(2)->t in M5 (proposition 4.1). The domain of n(3) -> t remains the same in both memories (proposition 4.2). We obtain the following memories:



Other deductions associated with the operator include the following rules:

- Any couple <loc,Var> existing in one of the two memories should also appear in the other memory (propositions 2 and 3);

- For all the pairs (<loc,V_0>,<loc,V_1>) in Tab(M_{in}) x Tab(M_{out}):

- If dom(V)∩dom(V_1)≠∅, the variable V cannot be stored at the location loc, so X ≠ loc (proposition 4.4)
- In other cases, we can deduce that dom(V_1) is included in dom(V_0) U dom(V) (proposition 4.5)

The solving process fails if the domain of X, or the domain of an abstract variable stored in M_{in} or M_{out}, or the domain of V becomes empty.

After constraint propagation, store_element succeeds if M_{in} is closed and the value pointed by X is known. Indeed, in this case all the information that permits to deduce the contents of Tab(M_{in}) and Tab(M_{out}) is available.

5.6. The load_element operator

The operator load_element(Tab(M),X,V) is added when the program loads an integer or pointer value from the memory at a given address. Loading a value does not modify the memory so it constraints only a single memory. Figure 12 illustrates the operator that maintains a relation between X, V and M, where pointer X points to a variable V in the corresponding *tableau* of M.

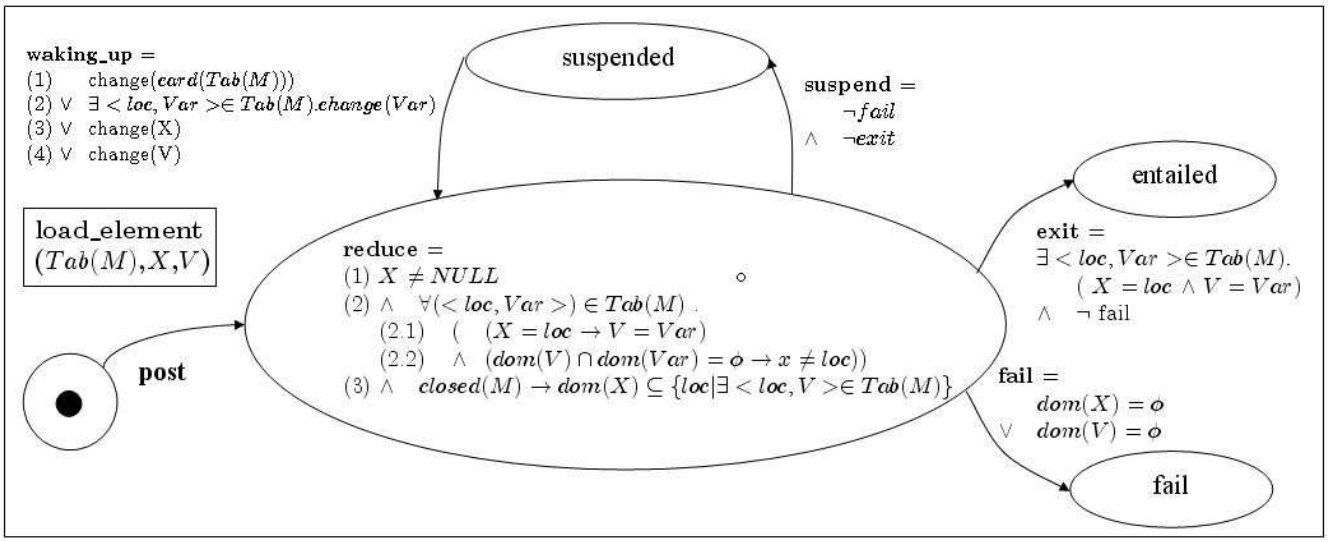


Figure 12. load_element operator

Proposition 2 expresses that for all the pairs <loc,V_{ar}> in Tab(M):

- If X points to loc, the variable V is loaded from the location loc and V=V_{ar}. The domain of V and V_{ar} is then dom(V)∩dom(V_{ar}).

- If $dom(V) \cap dom(V_{ar}) \neq \emptyset$, the variable V cannot be loaded from the loc location and then $X \neq loc$.

The solving process fails if the domain of X or V becomes empty, while it succeeds if there is a pair $<loc, V_{ar}>$ in Tab(M) such that X=loc and $V=V_{ar}$.

6. Experimental results on the Josephus program

We implemented the operators that are described above. Our system is able to take a program written in a restricted syntax of the C language and generates automatically test data w.r.t. some testing objectives. The system is developed in C and Prolog and follows the principles of the previous implementation INKA.

As an illustration of the efficiency of our operators, we generated test data for the Josephus program in figure 1. We considered several testing objectives. Among them, we generated a test suite that covers all the branches of the program in less than 1 msec of CPU time. The results were computed on an Intel Pentium, 2.16 GHz machine running Windows XP with 2.0 GB of RAM. To cover this objective, INKA first tries to find a test case to reach the deepest instructions of the control flow graph. In the case of the Josephus, it means that it tries to enroll the loop **while3** at least once. The test suite generated contains {(3,2)} as values for m and n. The output memory shape obtained with our model is in accordance with the one obtained by executing the program, which shows that the operators faithfully model dynamic memory management.

We also dealt with more complex requests as the one or reaching four iterations of **while3** in the first iteration of **while2**. It is worth noticing that reaching this testing objective is hard, as witnessed by the fact that a random test data generator will probably never achieve to reach it. In fact, the likelyhood of drawing a test data (values for n and m) such that this objective would be covered is not far from zero, as there is only a single value for m able to satisfy it. Thanks to our constraint reasonning on operators, we obtained the test case m=5, n=2 in 109 msec. If we take into account the wide domain for m and n as input values, the probability to reach this objective with random test case generation is low.

Regarding the objective of unrolling k times the loop **while2**, which is the objective described in introduction of this paper, we obtained the results shown in the curve on figure 13. The test cases obtained are of the form m=0, n=k+1. When k is less than 15, the generation of a test case to reach the objective takes less than 5 sec. We claim that these results are promising because they confirm the high deductive potential of our approach. Nevertheless, as shown in the curve, runtime increases exponentially with the value of k. Indeed, the number of operators to handle dynamic memory management in the constraint system increases with k. So the number of constraint waking up, costly in time, increases with the number of operators.

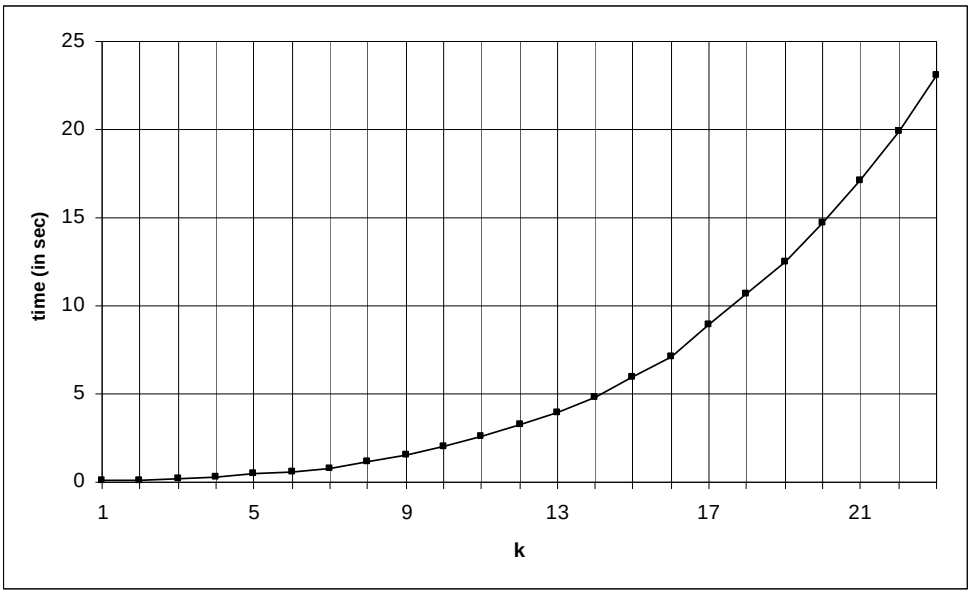


Figure 13. runtime for the generation of a test case reaching k iterations of while2 in the Josephus program

7. Related Works

Dynamic test data generation exploits program executions to find test data that satisfy a testing objective [20]. In this process, program executions guide the search towards the next test data by optimizing a cost function called the branch function. As the search is only based on concrete executions, this process handles dynamic data structures. Visvanathan and Gupta studied in [21] the problem of generating test data for functions with pointer inputs. They proposed to enumerate the possible data structures shapes by exploiting an address table symbolically computed during the execution of a program path. Sai-ngern et al. [22] followed a similar path by managing the table with a dynamic linear array. Unlike these approaches, our method is symbolic and it does not depend on execution trials. Symbolic test data generation is usually considered as more powerful to deduct information and our operators that model dynamic memory management are equipped with deduction rules in order to prune early the search space of possible about data structures shapes.

Williams et al. [9] and Sen et al. [11] address the problem of generating test data for C functions with dynamic structures by using symbolic execution and constraint solving techniques. In their approaches, constraints on input values permit to handle pointer relationships and aliasing problems occur only within input data structures. In our approach, we have proposed dynamic memory management operators able also to deal with pointer aliasing problems that are located in the source code. PathCrawler [9] and CUTE [11] are two test data generators trying to cover all the feasible paths of C programs. Both systems try to solve path conditions in order to find the next test data that will follow a path that improve the current coverage of the program. These tools are path-oriented, meaning that they require a path to be selected first. Unlike path-oriented and among other advantages, goal-oriented methods

such as the one presented in this paper, exploit the early detection of non-feasible paths to prune the search space made up of all the paths that reach a given branch. Considering all paths that reach a given branch is usually unreasonable as the number of control flow paths can be exponential on the number of decisions of the program or even infinite when loops are unbounded. Moreover, thanks to the constraint reasoning on operators, our implementation permits to express more complex testing objectives such as reaching a selected point at certain iterations of a while loop. This is particularly interesting for programs that build dynamic data structures as such requests help verifying their shapes during testing. However, one disadvantage of our approach is that it requires the constraint solver to be adapted and modified which prevents the usage of some commercial solvers that are sometimes more efficient.

8. Conclusion and perspectives

In this paper, we presented a new constraint-based model that handles dynamically allocated data structures in goal-oriented test data generation. Our model is built over specific constraint operators that are equipped with powerful deduction rules. These operators handle dynamic memory allocation, deallocation, loading and update of pointed structures. We implemented these operators within INKA a test data generator for programs written in a restricted subset of C. This implementation was challenging because it required building a new constraint solver over pointer and memory variables. However, our implementation still suffers from some restrictions. It is based on a memory model that does not include physical information about variables. For example, size of data types and bit vectors are not considered in the model and then, data structures such as unions or bit fields and physical type casting cannot be handled. Function pointers have also been left apart for the moment. Our goal-oriented approach is based on testing objectives that specify statements or branches in structured programs only and then goto statements are not currently handled. For all these reasons, we consider that the subset of the C language currently handled by our tool is too tight to be practically useful on real-world programs. But before evaluating our model on larger programs, we wanted to be sure that the approach was suitable and efficient on small but complex programs. Thanks to our constraint model, we successfully generated test data that cover complex testing objectives for the C program Josephus, which involves the creation and destruction of circular linked lists.

Our future work will be dedicated to extend this approach to inter-procedural test data generation and to a larger subset of the C language. Dealing with function calls and dynamic memory allocation requires paying attention on how constraint systems are built as the number of constraints can grow exponentially with the number of function calls. Hence, just inlining function calls will not be an acceptable solution and we would probably need some kind of abstractions. Dealing with unstructured code will also be a real challeng as our approach builds over constraint operators that model control structures (conditionals, loops) and goto statement usually break the flow. In fact, modelling exactly the semantics of such constructions is difficult and our line of work will be focussed on the possible combination of constraints and abstractions to approximate the behaviours of these statements in constraint-based automatic test data generation.

9. References

[1] DeMillo, R. A. and Offutt, A. J. 1991. "*Constraint-Based Automatic Test Data Generation*". IEEE Trans. Softw. Eng. 17, 9 (Sep. 1991), 900-910.

[2] Jeremy Dick and Alain Faivre "*Automating the Generation and Sequencing of Test Cases from Model-Based Specifications*" *Proc. Of Formal Methods Europe* (FME 1993), pages 268-284

[3] Bruno Marre, "*Toward Automatic Test Data Set Selection Using Algebraic Specifications and Logic Programming*" *in Proc. of the Int. Conf. on Logic Programming* (ICLP 1991), pages 202-219

[4] Gotlieb, A. , Botella, B. and Rueher, M., *"Automatic Test Data Generation Using Constraint Solving Techniques"*, *in Proc. Of the Int. Symposium on Software Testing and Analysis* (ISSTA 1998), Clearwater Beach, FL, USA, 1998

[5] Gotlieb, A. and Denmat, T. and Botella, B., *"Goal-oriented test data generation for programs with pointer variables"*, *in Proc. of the 29th IEEE Annual International Computer Software and Applications Conference* (COMPSAC 2005), Edinburh, Scotland, 2005, pp. 449-454

[6] Botella, B. and Gotlieb, A. and Michel, C., *"Symbolic execution of floating-point computations"*, The Software Testing, Verification and Reliability journal 16 (2), John Wiley, 2006, pp 97-121

[7] Gotlieb, A. and Botella, B. and Watel, M., *"Inka: Ten years after the first ideas"*, *in Proc. of the 19th International Conference on Software and Systems Engineering and their Applications* (ICSSEA 2006), Paris, France, 2006

[8] Meudec, C., "*ATGen: automatic test data generation using constraint logic programming and symbolic execution"*, The Software Testing, Verification and Reliability journal 11 (2), John Wiley, 2001

[9] Williams, N., Marre, B., Mouy P. and Roger, M. *"PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis" In Proc. of the 5th European Dependable Computing Conference* (EDCC 2005), Budapest, Hungary, LNCS Vol. 3463/2005, Springer-Verlag, 2005, pp 281-292

[10] Godefroid, P., Klarlund, N., and Sen, K. 2005. *"DART: directed automated random testing"*. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI 2005), Chicago, IL. pp 213-223

[11] Koushik Sen and Darko Marinov and Gul Agha, *"CUTE: a concolic unit testing engine for C"*, *In Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering* (ESEC/FSE-13), ACM Press, Lisbon, Portugal, 2005,pp.263-272

[12] D. Lewin and L. Fournier and M. Levinger and E. Roytman and G. Shurek, *"Constraint Satisfaction for Test Program Generation"*, *in Proc. Of the IEEE International Phoenix Conference on Communication and Computers*, (IPCCC 1995) Phoenix, 1995

[13] Bin, E. and Emek, R. and Shurek, G. and Ziv, A., *"Using a constraint satisfaction formulation and solution techniques for random test program generation"*, IBM Systems Journal 41 (3), 2002

[14] Daniel Jackson and Mandana Vaziri, *"Finding bugs with a constraint solver"*, *In Proc. of the ACM SIGSOFT Iinternational Symposium on Software Testing and Analysis* (ISSTA 2000), pp. 14-25, Portland, Oregon, United States, 2000

[15] C. Pasareanu, M. Dwyer and W. Visser, *"Finding Feasible Abstract Counter-Examples",* STTT Journal, 5, (1),2003

[16] Collavizza, H. and Rueher, M., "*Exploration of the Capabilities of Constraint Programming for Software Verification"*, *In Proc. of the Conf. on Tools and Algorithms for the Construction and Analysis of Systems* (TACAS'06), Vienna, Austria, 2006

[17] Ferguson, R. and Korel, B. 1996. *The chaining approach for software test data generation*. ACM Transactions On Software Eng. And Methodology. 5, 1 (Jan. 1996), 63-86

[18] Sedgewick, R. "*Algorithms in C"*, Addison-Westley publishing company, 1988

[19] Utting, M. and Legeard, B. "*Practical Model-Based Testing: A Tools Approach*" Morgan Kaufmann, 2007

[20] Korel, B. "*Automated Software Test Data Generation*," IEEE Transactions on Software Engineering ,vol. 16, no. 8, pp. 870-879, August, 1990.

[21] Visvanathan, S., Gupta, N. "*Generating Test Data for Functions with Pointer Inputs,*" *in Proc. Of the 17th IEEE Automated Software Engineering Conference (ASE'2002)*, pp. 149-160, 2002.

[22] Sai-ngern, S. and Lursinap, C. and Sophatsathit, P., *"An address mapping approach for test data generation of dynamic linked structures",* Information and Software Technology (47), 2005,pp. 199-214